Implementing the MyFEM Embedded Domain-specific Language

Jonathan Riehl

Implementing the MyFEM Embedded Domain-specific Language

Jonathan Riehl
University of Chicago
jriehl@cs.uchicago.edu

Abstract

Mython, an extensible variant of the Python programming language, exposes the tools and libraries of its implementation to users. This kind of reflection allows domain-specific language implementations to reuse parts of Mython. This paper looks at the MyFEM domain-specific language, showing how MyFEM reuses the tools of Mython. MyFEM is a language for describing partial-differential equations and boundary conditions as a part of the finite element method. Using MyFEM as a high-level interface, users can generate fast, scalable scientific codes using very compact source programs. This paper also discusses plans for further automation, where MyFEM and Mython act as a seamless front end. These plans involve not just hiding the target code from the user, but also any necessary build details, including linkage and foreign function interfaces.

1. Introduction

Domain abstractions are useful for hiding the complexity of their implementation from users. Ideally, domain abstractions lower both the development and maintenance costs of creating domain-specific programs. Optimization of domain-specific code often works against abstraction by forcing users to work with the low-level implementation code instead of the high-level abstraction layer. Reliability can add further complexity to optimization through of static and run-time checks. The result is a tradeoff between reliability, maintainability, and optimality.

The FEniCS project (FEniCS 2008), seeks to develop more reliable, maintainable, and optimal scientific software. The FEniCS project serves this goal by providing a suite of software for building scientific simulations. Their approach to improving simulations is automated code generation (Terrel et al. 2008), and they are pursuing technologies that can help generate glue code for their existing suite of tools.

The Mython programming language (Riehl 2008a) presents a good candidate for a glue platform because of its extensible front-end, back-end, and the native extensibility of Python's runtime. The front-end extensibility of Mython allows customized concrete syntax. Users can use Mython's back-end extensibility to extend the code generator, adding domain-specific optimizations. The Mython implementation consists of a compiler that generates Python modules, meaning Mython programs can benefit from the native extensibility of the Python runtime. For example, Python bindings are available for many of the FEniCS libraries.

Figure 1 presents an example of Mython and the FEniCS embedded domain-specific language, MyFEM. This program generates a C++ function that is 64 lines of code, with multiple nested loops that are up to six deep. The actual domain-specific code appears on lines six through nine, showing that MyFEM can provide an order of magnitude reduction in line count. This is not atypical in the finite element method domain. Input and output sizes for the related FEniCS form compiler (FFC) can differ by four orders of magnitude, with several thousand lines of code generated from a six line input (Kirby and Logg 2006).

This paper describes the purpose and implementation of MyFEM. It serves as a case study of embedded domain-specific language implementation in an extensible host language. Section 2 reviews background material, including a brief review of the finite element method and the Mython programming language. Section 3 describes the MyFEM compiler design and how it is used to extend Mython. This paper concludes by identifying open issues and related work.

2. Background

This work is a fusion of two disciplines in computer science: scientific computing and programming languages. This section provides some background on scientific computing and embedded domain-specific language implementation. It begins with an overview of the motivations and goals of automating scientific computing. It then describes the finite element method, a computational tool for simulation. Finally, it provides an introduction to the Mython programming language, which uses compile-time metaprogramming to embed domain-specific languages as concrete syntax.

2.1 Automating Scientific Computing

Simulation grants scientists and engineers the ability to do experiments that are not feasible in the real world. Some experiments are not feasible because of resource constraints, and include such domains as high energy physics. Other experiments may be prohibitive because they pose real risks to the experimenters, their environment, or the public. For this very reason, one of the first and still common applications of the digital computer is the simulation of atomic physics. Even when resources are available and sufficient precaution can be taken, researchers can use simulation as a tool for guiding and validating both theory and experimentation.

In simulations people input mathematical models of systems and a set of boundary conditions, which may include initial conditions. Frequently, these models employ partial-differential equations (PDE's) to relate the change in the system as a function of the system's instantaneous state (Logg 2004). An example of this might be an equation of the form:

$$u' = f(u)$$

In this model, the function f presents a means of calculating an instantaneous rate of change of a state vector, u', using only the current state vector u. Simulation involves the calculation some unknown, which can either be u' (which is trivial in this case), u, or f.

One of the most important aspects of automating scientific computing is the management of error. Solving problems using digital simulation necessitates discretization of the problem domain. Since many of these models are continuous, discretization introduces error into the calculations. These calculations are often then repeated

```
 1: from basil.lang.fenics import FEniCS
 2: from basil.lang.fenics.bvpir import *
 3: quote [myfront]: from basil.lang.fenics import FEniCS
 4:
 5: quote [FEniCS.bvpFrontEnd] Laplace:
 6:    TestFunction v
 7:    UnknownField u
 8:
 9:    <grad v, grad u>
10:
11: print "void Jac_Laplace (const Obj<ALE::Mesh>& m,"
12: print "                  const Obj<ALE::Mesh::real_section_type>& s,"
13: print "                  Mat A, void * ctx)"
14: print FEniCS.bvpIRToCplus(Laplace)
```

Figure 1. An example MyFEM code generation program.

by the simulation, which can cause the error to compound. This compounding nature of error can limit a simulation's usefulness.

Iterative methods allow management of a simulation's error, but they also impact the simulation's portability and maintainability. Iterative methods track error, iterating computations to decrease error when the error exceeds expectations. This iteration can add both space and time complexity to the underlying program. Therefore, automation of scientific computing includes a quantification of acceptable error as an input, in addition to a model and boundary conditions.

2.2 The Finite Element Method

The finite element method (FEM) is a means of discretization for the purpose of solving partial-differential equations. Typically the finite element method, particularly its application to boundary value problems, has three inputs: a mesh, boundary conditions, and partial differential equations (PDE's) that govern the relationship between field values. A mesh is typically a discretization of space into a triangulation, but this abstracts to arbitrary manifolds. The discretization also determines the relationship between field values and vertices in the mesh.

The MyFEM implementation allows engineers and scientists to specify boundary and conditions and PDE's, generating C++ code that interfaces with the Sieve (Knepley and Karpeev 2008) and PETSc (Balay et al. 2006) libraries. The Sieve library handles user mesh inputs. Sieve lets users represent arbitrary meshes, both in dimension and shape. This representation allows MyFEM to emit integration loops independent of the mesh, the finite element, and the solver. The PETSc library provides a variety of solvers which are used to solve the algebraic systems that result from discretization.

MyFEM's role in the finite element method is the specification of partial differential equations and boundary conditions. For example, an engineer may want to find a field u where given some test function v, the following constraint holds [1]:

$$\int_{\Omega} \nabla v \cdot \nabla u - \int_{\Omega} vf = 0 \tag{1}$$

The domain of integration, Ω, is explicitly given above. This domain corresponds to a required input of a MyFEM program, and left implicit in the MyFEM source. The MyFEM syntax for expressing equation (1) is:

```
TestFunction v
UnknownField u
CoordinateFunction f

<grad v, grad u> - <v, f>
```

This example illustrates how MyFEM uses inner-product notation (`<,>`) to denote integration over the product of two subexpressions. The result is a concrete syntax that is closer to actual mathematical notation, while still being machine readable.

2.3 Mython: An Extensible Language

Mython (Riehl 2008a) is an extensible variant of the Python language (van Rossum 2006). Mython achieves extensibility at all levels of language implementation, allowing developers the ability to change the language syntax, semantics and runtime from inside the language itself. Mython allows easy incorporation of new syntax through interoperability with popular parser generators. Compile-time reflection allows developers to easily embed and extend the syntax of Mython and its embedded domain-specific languages. Mython achieves semantic extensibility by reflecting the compiler to the Mython language. This allows developers to do such things as add optional static typing to increase program safety, and implement optimizations that increase program performance. Mython inherits runtime extensibility from the Python language itself, affording both dynamic runtime extensions via multiple third-party foreign function interface generators (Beazley 1996), and modifiable runtime semantics via Python meta-classes. Mython achieves all these features by starting with a popular programming language, Python, and adding open compiler technology (Tatsubori et al. 2000), staged computation (Taha 2003), and compile-time metaprogramming (Tratt 2005). MyFEM uses all these features to create a programmable interface to scientific computing automation that is both flexible and fast.

The example in Figure 1 illustrates several of the properties discussed above. The Mython compiler evaluates `quote` statements at compile time. The expression in square braces is evaluated first, and the result of that evaluation is used to evaluate the nested code. The compiler has its own environment for name binding. In the example, line three extends the compiler by importing the FEniCS language library. The compiler binds the `myfront` name before compilation to a function that evaluates nested Python code in the compilation environment. Line three may seem redundant because of the import on line one. However, line one only binds the name at run time, requiring the second, compile-time import. Line five uses a front-end function from the MyFEM library to parse the embedded MyFEM code on lines six through nine. The result generates code that is bound to the `Laplace` identifier at run time.

[1] This example, and how a user derives this constraint, are given in § 2.2 of Gockenbach (Gockenbach 2006)

Line fourteen uses the `Laplace` identifier at run time to generate C++ code.

Syntactic and semantic details that are not directly related to MyFEM's implementation are outside the scope of this paper. The author's dissertation (Riehl 2008b) provides these details for both the Mython and MyFEM languages.

3. Implementing MyFEM

This section describes the implementation of MyFEM, and how this implementation interoperates with Mython. The global design of MyFEM is typical for a compiler, where a front-end handles translation of the concrete syntax into some intermediate representation, and a back-end manages optimization and code generation for the intermediate representation. Section 3.1 looks at the general methods used to embed a domain-specific language in Mython. Section 3.2 covers the front-end implementation details of MyFEM. Finally, section 3.3 discusses the code optimization and generation portions of the language implementation.

3.1 Methodology

The most straightforward means of embedding a formal language in Mython is to build a parser using a parser generator. The person developing the embedding creates a Python function that parses a string, and outputs a parse tree. The Python function can either use a native parser from a Python parser generator, or a wrapped foreign function. Mython includes a function, `myescape()`, that will generate abstract syntax from a Python value. The compiler passes the resulting abstract syntax to the code generator, causing the compile-time value to be rebuilt at run time. Binding and using a composition of the parser and escape function at compile time, the Mython compiler will parse embedded code and bind the result at run time.

One problem with simply wrapping a parser is the verbosity of the resulting concrete parse tree. Therefore, it is often convenient to transform the concrete parse tree into an intermediate representation that is more closely representative of the underlying language semantics. Abstract syntax trees are one such intermediate representation, but developers can find creating abstract syntax tree constructors and utilities tedious due to a duplication of a lot of boilerplate code. The following subsections identify how Mython and its libraries help developers generate parsers, traverse trees, and generate abstract syntax trees.

3.1.1 Parsing

Part of Mython's reflective nature involves exposing its parser generator to the surface language. The `pgen` parser generator is an LL(1) parser generator that takes extended Backus-Naur form (EBNF) grammars and generates parsing automata. Originally developed in C by Guido van Rossum for the implementation of Python, Mython offers a reimplementation of `pgen` in Python. Mython uses this version of `pgen` to generate parsers for Python, Mython and MyFEM.

This approach has two drawbacks. First, `pgen` has no integration with its lexical syntax. Parsers generated by `pgen` are constrained to using either Python's lexer or some adaptive lexer that outputs Python-compatible lexical tokens. This limitation does not pose too great a hindrance, since keywords are not given specific lexical classes in Python, but passed as identifiers to the parser. This also means that the formal language is whitespace sensitive, since Python uses indentation for lexical scoping. MyFEM uses keywords for several of its new operators, sidestepping the issue of limited lexical extensibility.

The second drawback involves the robustness of `pgen`. Its heritage as a Python-specific tool mean it has not been hardened for general-purpose parser generation. It does not provide good error detection for ambiguity, and grammars that are not in the family of LL(1) languages. Finally, `pgen` does not include support for error recovery in the generated parsers. Both `pgen` and the parsers it generates will halt upon detection of the first syntax error.

The output of `pgen` generated parsers is a concrete syntax tree. These trees have the following structure:

$$\begin{array}{rcl} node & := & (payload, children) \\ payload & := & ID \\ & | & token \\ token & := & (ID, string, lineno, \ldots) \\ children & := & [node_1, \ldots, node_n] \end{array}$$

3.1.2 The Visitor Pattern in Mython

Mython's compiler uses an implementation of the visitor pattern (Gamma et al. 1995) to walk and transform tree data structures. The approach described by this section closely follows other implementations in Python, including one found in Python `compiler` library (van Rossum 2008). Mython reflects this implementation up to the surface language as an abstract base class, `Handler`. Mython uses subclasses of the `Handler` class to both transform concrete syntax trees to abstract syntax tress, as well as generate code from abstract syntax trees.

The `Handler` class implements a generic visitation method, `handle_node()`. The `handle_node()` method first looks at the current tree node being visited, and uses the `get_handler_name()` method to determine a method name that specifically handles the given node type (by default, this is "`handle_`*NT*`()`", where *NT* is the name of the nonterminal symbol for the node). The `handle_node()` method then uses introspection to see if the handler method exists in the current `Handler` instance. If the specific method exists, `handle_node()` calls it, otherwise it calls a generic handler method, `handle_default()`.

Mython users can follow the same pattern as Mython itself, specializing the `Handler` class to walk both concrete and abstract syntax trees for domain-specific languages. This requires the user to specialize several methods. These include the `get_nonterminal()`, which `get_handler_name()` uses to generate the dispatch name, `get_children()`, which the visitor uses to determine a node's children, and finally `handle_default()`, which is the default handler method.

3.1.3 Abstract Syntax

Mython reflects the abstract syntax definition language (ASDL) to its surface language (Wang et al. 1997). ASDL is a domain-specific language for describing abstract syntax data structures. Mython inherits its use of ASDL from Python, where both languages use ASDL to define their abstract syntax. While not part of the standard library, Python implements an ASDL parser and a utility for translating ASDL to C. Mython builds on the Python utility by providing a tool that translates from ASDL to a Python module. The module defines a class hierarchy for representing abstract syntax. Users can in turn use introspection of the output Python modules to create pretty-printers, visitors and other utilities.

Mython has a second visitor base class, `GenericASTHandler`, that is designed specifically for ADSL syntax trees. The `GenericASTHandler` class simplifies the number of methods needed to implement the visitor pattern, directly using a node's type name to determine its handler method, and providing a default handler that performs a depth-first traversal of the tree. The Mython compiler specializes this class into several subclasses, including one for lexical scoping calculations, and one for code generation.

3.2 The Front-end

The MyFEM front-end applies the methods described in section 3.1 to create a front-end function that Mython users can use to embed MyFEM programs. The example code in figure 1 shows such an embedding, where line 3 imports the MyFEM language definition module into the compiler. Line 5 then uses the `FEniCS.bvpFrontEnd` function to parse the MyFEM code on lines 6–9.

The `bvpFrontEnd()` function is a composition of several functions:

```
def bvpFrontEnd (name, text, env):
    global bvpParser, bvpToIR, bvp_escaper
    cst, env_1 = bvpParser(text, env)
    ir = bvpCSTToIR(cst)
    esc_ir = bvp_escaper(ir)
    stmt_lst = [ast.Assign([ast.Name(name,
                  ast.Store())], esc_ir)]
    return stmt_lst, env_1
```

The first function, `bvpParser()`, is a parser generated using the `pgen` tool. This outputs a parse tree. The front-end then sends the parse tree to a transformer function, `bvpCSTToIR()`. This function translates the parse tree into several intermediate representations, finally resulting in an abstract syntax tree for an imperative intermediate representation. The imperative abstract syntax tree is translated into Python abstract syntax by the `bvp_escaper()` function, completing the quotation.

3.3 The Back-end

A back-end is responsible for performing any user specified optimizations and generating target code from the intermediate representation. Users can implement domain-specific optimizations as transformations of abstract syntax using the AST visitor techniques described in section 3.1.3. A user defines a custom back-end by composing his or her optimizers with a code generator. This section describes a simple back-end that generates C++ target code from the intermediate representation output by the front-end documented in the previous section.

The example in figure 1 uses the front-end to generate the abstract syntax tree for an imperative intermediate language. As part of the Mython quotation mechanism, the Mython program reconstructs this tree at run-time and binds it to the `Laplace` identifier. The script then passes the reconstructed tree to the back-end function, `bvpIRToCplus()` (in the `FEniCS` module), on line 14. The back-end function constructs a visitor for the intermediate representation, and uses the visitor object to generate a string containing the target C++ code.

MyFEM uses ASDL to define the imperative intermediate language. MyFEM follows the ASDL AST visitation methodology, specializing the `GenericASTHandler` class. The resulting subclass, `IRToCplusHandler`, contains custom handler methods responsible for translating each of the abstract syntax nodes into C++ strings.

4. Open Issues

MyFEM is a work in progress, presenting several remaining implementation challenges. These challenges include domain-specific optimizations, type systems, and making MyFEM and Mython simpler to use.

4.1 Domain-specific Optimization

Mython's extensibility allows users to create domain-specific optimizations that can interface with the Mython compiler's code generator. We are looking at adding GHC-style rewrites (Peyton Jones et al. 2001) to the Mython language as one method of presenting users with an abstraction that hides details of the Mython compiler's implementation. Rewrites pose problems in Mython because non-local names are neither bound at compile nor import time. Naive approaches to rewrite implementation would restrict them to rewriting specific, locally bound names, or expression syntax that deals with known types (such as list comprehensions). MyFEM also has a set of domain-specific optimizations that can not be represented using rewrites. The next subsection discusses a specific example of one such optimization.

4.2 Type Systems

MyFEM includes a type system that only ensures that a MyFEM program is capable of calculating a function for the residual (where the residual is a scalar that is suggestive of the error of an approximate solution). One possible optimization in MyFEM involves selecting a solver known to have good run-time performance for determining the requested residual. Specifically, MyFEM constraints can be linear, non-linear, or a combination of both linear and non-linear terms. Separating terms based on their linearity, and using separate solvers for specific sub-terms presents an opportunity for improving the run-time performance of MyFEM programs. We are currently looking at extending the current type system to account for the linearity of terms. Alternatively, linearity might be easier to express as a separate type system, since it has little impact on correctness concerns.

4.3 Gluing It All Together

At the time of writing, the MyFEM implementation's back-end only generates C++ code. Users must still compile and link their MyFEM programs in a separate compilation pass. One of the goals behind Mython is to provide an environment where the lines between compiler-compile time, compile time, and run time are blurred, if not invisible to the user. In order to accomplish this, we plan to add a just-in-time compilation suite. Additionally, Mython should be able to generate glue code on the fly, providing a just-in-time foreign function interface generator.

The current division of MyFEM translation between compile time and run time does not provide any specific benefit (other than providing syntactic checks at compile time). This division suits Mython's use of staging as a means of controlling and containing extensibility, and serves as a test of the Mython implementation. Other possible targets for the current MyFEM compiler are embedding the generated C++ code as a string or as constructors for some lower-level intermediate representation. These embeddings lose domain-specific information if a user wants to apply transformations at run time.

5. Related Work

The MyFEM language is related to several other scientific computing systems which inhabit similar roles in the FEniCS project. The FEniCS form compiler, or FFC, is a Python based compiler that translates from Python code to C++ (Kirby and Logg 2006). SyFi is a C++ library for building finite elements (Alnæs and Mardal 2007). SyFi uses the GiNaC symbolic computation library to express partial-differential equations (Alnæs and Mardal 2007; Vollinga 2006). SyFi still requires that users write code to assemble the matrices used in simulation, where MyFEM will automate this process.

The Mython host language is related to a set of other extensible languages, most notably Converge (Tratt 2005). Though developed independently, both Mython and Converge are quite similar. Converge features a Python-like syntax, and both languages feature special syntax that distinguishes code evaluated at compile time

from run-time code. Unlike Mython, Converge includes expression syntax for using compile-time metaprogramming, and binds run-time code in the compile-time environment.

The Fortress compiler also allows user-defined syntax (Allen et al. 2006). Users add syntax by defining parsing functions, and then associate a custom pair of delimiters with each function. The compiler pre-processes embedded code using these parsing functions, which return Fortress abstract syntax. Unlike Mython and Converge, parsing functions can not appear in the same module, requiring separate compilation of the extension syntax before its use.

Many extensible languages should allow users to define and use custom syntax with only a few additional lines of code. These languages start with a general purpose language, and allow users to embed syntax definition languages. The OMeta language inverts this approach, starting with a syntax definition and transformation language, and allowing users to embed and extend other languages (Warth and Piumarta 2007).

Stratego/XT and the MetaBorg pattern inspired the design and language embedding approach of Mython (Bravenboer and Visser 2004). Mython uses compile-time metaprogramming to realize the ideas and solve some issues with a previous proposal for building extensible languages (Riehl 2006). Future work slated for Mython includes embedding the Stratego and SDF languages. As the Mython language implementation story evolves, the MyFEM implementation presents an opportunity for comparative analysis between visitor and strategy methods.

6. Conclusions

This paper presented a case study of using an extensible language to embed another domain-specific language. This work differs from previous work in the scientific computing community by providing both concrete syntax and the opportunity for user-level domain-specific optimization. This work also provides a case study of using a recent class of extensible languages. These languages allow developers to embedded languages with concrete syntax and permit integration of some or all of their compiler with the host language compiler.

Acknowledgments

The author wishes to thank Matthew Knepley, John Reppy, and Andy Terrel for their help and feedback.

References

Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress Language Specification, Version 1.0α. Technical report, Sun Microsystems Inc., September 2006.

Martin Alnæs and Kent-Andre Mardal. SyFi User Manual, September 2007. Available at `http://www.fenics.org/wiki/Documentation`.

Satish Balay, Kris Buschelman, Victor Eijkhout, Will iam D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.3.2, Argonne National Laboratory, September 2006. URL `http://www.mcs.anl.gov/petsc/docs`.

David M. Beazley. SWIG: An Easy-to-Use Tool for Integrating Scripting Languages with C and C++. In *Proceedings of the Fourth USENIX Tcl/Tk Workshop*. USENIX Assoc., 1996.

Martin Bravenboer and Eelco Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In Douglas C. Schmidt, editor, *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04)*, pages 365–383, Vancouver, Canada, October 2004. ACM Press.

FEniCS. FEniCS Project, 2008. `http://www.fenics.org/`.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995. ISBN 0201633612.

Mark S. Gockenbach. *Understanding and Implementing the Finite Element Method*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006.

Robert C. Kirby and Anders Logg. A compiler for variational forms. *ACM Trans. Math. Softw.*, 32(3):417–444, 2006. ISSN 0098-3500. doi: http://doi.acm.org/10.1145/1163641.1163644.

Matthew G. Knepley and Dmitry A. Karpeev. Mesh Algorithms for PDE with Sieve I: Mesh Distribution. *Scientific Programming*, 2008. (To appear).

Anders Logg. *Automation of Computational Mathematical Modeling*. PhD thesis, Chalmers University of Technology, 2004.

Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In Ralf Hinze, editor, *Preliminary Proceedings of the 2001 ACM SIGPLAN Haskell Workshop (HW '01)*, pages 203–233, Firenze, Italy, September 2001.

Jonathan Riehl. The Mython Language, 2008a. Available at `http://www.mython.org/`.

Jonathan Riehl. Assimilating MetaBorg: Embedding language tools in languages. In *Proceedings of the Fifth International Conference on Generative Programming and Component Engineering (GPCE'06)*, October 2006.

Jonathan Riehl. *Reflective Techniques in Extensible Languages*. PhD thesis, University of Chicago, 2008b.

Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50, 2003.

Michiaki Tatsubori, Shigeru Chiba, Kozo Itano, and Marc-Olivier Killijian. OpenJava: A class-based macro system for java. In *Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering*, pages 117–133, London, UK, 2000. Springer-Verlag. ISBN 3-540-67761-5.

Andy R. Terrel, L. Ridgway Scott, Matthew G. Knepley, and Robert C. Kirby. Automated FEM Discretizations for the Stokes Equation. *BIT Numerical Mathematics*, 2008. *(To appear.)*.

Laurence Tratt. Compile-time meta-programming in a dynamically typed OO language. In *Proceedings of the Dynamic Languages Symposium*, pages 49–64, October 2005.

Guido van Rossum. Python Reference Manual (2.5), September 2006. Available at `http://www.python.org/doc/2.5/ref/ref.html`.

Guido van Rossum. Python Library Reference (2.5.2), February 2008. Available at `http://www.python.org/doc/2.5.2/lib/lib.html`.

Jens Vollinga. GiNaC: Symbolic computation with C++. *Nucl. Instrum. Meth.*, A559:282–284, 2006.

Daniel C. Wang, Andrew W. Appel, Jeff L. Korn, and Chris S. Serra. The Zephyr Abstract Syntax Description Language. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, pages 213–228, October 1997.

Alessandro Warth and Ian Piumarta. OMeta: an Object-Oriented Language for Pattern Matching. In *Proceedings of Dynamic Languages Symposium '07*, October 2007.